# Ex. 2, Matrices and Handles

Stefan Karlsson

February 17, 2014

## 1 Matrices

Matrices are easily created, such as the following 3 *different* ones

```
a = [8 -10  15  1.1  pi sqrt(2)]        %1 row , 6 columns (a vector)
A = [8 -10  15; 1.1  pi sqrt(2)]        %2 rows, 3 columns
B = [8 -10; 15  1.1; pi sqrt(2)]        %3 rows, 2 columns
```

Or as:

```
A = [8    -10  15     ;...
     1.1  pi   sqrt(2)]

B = [8   -10    ;...
     15  1.1    ;...
     pi  sqrt(2)]
```

Inconsistent input will generate errors:

```
Bad = [8 -10  15  ;...
       1          ]  %will generate error
```

Easiest way to generate simple matrices are by commands such as:

```
zeros([2, 3])         %all zeros
 ones([2, 3])         %all ones
 rand([2, 3])         %all random values between 1 and 0
  eye([2, 3])         %all 1 along diagonal, 0 elsewhere
```

Matrices are used in countless applications. For now, we will think of matrices simply as tables of real-valued scalars[1], which we can easily visualize as grayscale images. Higher value in the matrix, will mean a brighter intensity of the pixel. In the example below, value 1 will give white pixels, and value 0 black ones:

```
A = eye(300);          %300 by 300 matrix with ones along diagonal
imagesc(A);            %show it as an image
colormap gray;         %use grayscale for visualization
axis image;            %best axis setting for image
```

[1] later, we will see applications into systems of equations, curve fitting and linear mappings

In this exercise, we will use `colormap gray` and `axis image` often. Therefore, a function `imagesc2` is provided that does this automatically. Feel free to look inside of `imagesc2.m` (although you are not required to). It consists of only a few lines, and is an example of how to do a so-called 'wrapper' function.

Images are matrices and we can load them into our workspace easily:

```
A = imread('eight.tif');                %loads a 242x308 matrix A
imagesc2(A);
```

With indexation we can modify images. Say we wish to make an image smaller, we can take every second pixel

```
A2 = A(1:2:end,1:2:end);
imagesc2(A2);
```

To make Matrices even smaller we can take every $n^{th}$ pixel

```
for n=1:30
  A2 = A(1:n:end,1:n:end);
  imagesc2(A2);
  title(['n = ' num2str(n)]);
  pause(1.5/n);
end
```

With indexation we can modify the image directly.

```
  %make a 100x100 white square
A(101:200,101:200)=255;      %gray value 255: white
imagesc2(A);

figure;
  %make a square of random values
randIm = rand(100)*255;       %100x100 image of random values
A(101:200,101:200)=randIm;
imagesc2(A);
```

We can have higher dimensional matrices as well. Color images usually contain 3 values per pixel (RGB - red, green and blue).

```
B = imread('ngc6543a.jpg');             %loads a 650x600x3 matrix B
imagesc2(B);
```

To reach the pixel values, we can use index `x`(coordinate), `y`(coordinate) and `c`(color) as `B(y,x,c)`. For example

```
B(20,10,1)          %The red component, at x=10 and y =20
B(20,10,:)          %the three RGB components
```

### The annoying order of x-y index

With the example above (`B(y,x,c)`), you may ask why it is NOT `B(x,y,c)`. We are used to x being listed first. This is an annoying source of bugs! Make sure you always check that x is the second index[2]

Going back to our RGB images, lets play around with the different channels:

```
  %indices for a 4x4 square box, from pixel position 2 until 5:
y = 2:5;
x = 2:5;
A = zeros([11,11,3]);          %make a matrix 11x11x3, of zeros

A(y   ,x   ,1) = 1;            %a square in the red channel
A(y   ,x+5 ,2) = 1;            %a square in the green channel
A(y+5 ,x+2 ,3) = 1;            %a square in the blue channel
imagesc2(A);
```

lets bring the boxes closer to each other, so they overlap:

```
y = 2:5;
x = 2:5;
A = zeros([11,11,3]);

A(y+1 ,x+1 ,1) = 1;
A(y+1 ,x+4 ,2) = 1;
A(y+4 ,x+2 ,3) = 1;
imagesc2(A);
```

In the resulting image, we see new colors emerging in the intersections of the boxes. One special color is white, which is defined as the color where R,G and B all reach maximum.

## 2 Handles

When you bring up a new figure, you use the function `figure`. Looking at the documentation of `figure`, we see additional uses involving 'handles'. A handle is a Matlab variable that is used to reference an object. Handles to figures are the most basic and simple ones. Lets get some figures up, and retrieve handles(after executing code, don't close the figures):

```
close all;           %kills all current figures
hFig1 = figure;
imagesc2(rand(10));  %display 10 x 10 random values

hFig2 = figure;
imagesc2(eye(10));   %display 10 x 10 identity matrix
```

[2]There is a reason why x is listed second. In linear algebra, the common notation is $A_{rc}$ where $r$ is the row, $c$ the column.

For several figures in existence, there is only one of them at a time that can be the 'CURRENT' figure. The current figure in Matlab is the figure that will take care of displaying stuff when you call functions such as plot or imagesc. If at any time, you select the window of a figure (e.g. clicking it) it will automatically become the current figure, the previous current figure loses the status. To make a specific figure active in code, we use `figure` again (this assumes you didnt close the figures from the previous code snippet):

```
figure(hFig1);
pause(2);
figure(hFig2);
```

To switch several times between them:

```
for n = 1:40
  pause(1/n);
  figure(hFig1);
  pause(1/n);
  figure(hFig2);
end
```

if you want to know which figure is the current one, use `gcf` (gets a handle to current figure):

```
if     gcf == hFig1
  disp('first figure is the current');
elseif gcf == hFig2
  disp('second figure is the current');
end
```

Figures have lots of different properties that we can manipulate. If you search for 'figure properties' in the documentation, you will find the exhaustive list (its huge). Among the properties, you can find 'Position'. Lets get the positions of our two figure windows:

```
pos1 = get(hFig1,'Position')
pos2 = get(hFig2,'Position')
```

Move one figure around manually(use the mouse), and verify that it changes the returned position vector. We use `get` to get any properties using handles, and `set` to set them. The last 2 elements of the position vector indicate the size of the figure, so we can modify that as follows:

```
pos1  = get(hFig1,'Position');              %get current position
figure(hFig1);                              %make the figure the current
for t = linspace(1,1.3,40);
  set(hFig1,'Position', [0 0 pos1(3:4)*t]);  %increase the widht and height a little
  pause(0.1);
end
  %move the figure back to original position:
set(hFig1,'Position', pos1);
```

## 2.1 Axes objects

The figure objects are not the objects we draw in directly. In each figure, there is at least one set of axes(x-y, and possibly z). The matlab 'axes object', is the object that is actually drawing things, and keeps track of the axes. When we call visualization functions, such as plot, it will draw in the current axes.

The function `subplot` generates axes objects. We have used `subplot` before, and like `figure` it also returns a handle. Leave the figures open after executing the following snippet:

```
close all;
t = linspace(0,2*pi,500);

hFig1 = figure;
%make 4 axes objects in the current figure
hAxes1 = subplot(2,2,1);
plot(sin(t));

hAxes2 = subplot(2,2,2);
plot(sin(2*t));

hAxes3 = subplot(2,2,3);
imagesc2(eye(10));

hAxes4 = subplot(2,2,4);
imagesc2(rand(10));
```

As with figure objects, we can move axes objects around and change their size[3]. Axes objects have other properties, that deal with how data is visualized. Some of these properties we have already modified, such as the limits of the axes. Using the function `ylim`, we can change the limits on the y-axis, for example. However, if we have 4 axes objects in our figure at the same time, how can we pick which axes for `ylim` to work on? All functions to manipulate axes work on the current axes by default. We can modify the property 'ylim' straight away using the handle:

```
set(hAxes1,'ylim', [-1,2])
set(hAxes2,'ylim', [-2,1])
```

As with figures, there is a huge list of properties that you can `set` and `get`.

## 2.2 Graphics objects

There is an internal hierarchy of visual objects in Matlab.

- On the top, there is the figure: the main container object, whose window you can move around.
- On the second level, there are axes objects inside each figure, which are the containers for all manner of visualizations of data and graphics.

[3]but only inside the figure they are created in

- Thirdly, inside each axes object, you can have several graphics objects, each with its own set of properties. We have lines, images and surfaces as examples.

The visualization function we have used the most is `plot`, and we recall the animation example from last exercise(Task 1, exercise 2):

```
t = linspace(-pi,+pi,700);
x = cos(t);
y = sin(2*t);

for p = 1:length(x)
   plot(x(p),y(p), 'o');
   xlim([-1.5,1.5]);
   ylim([-1.5,1.5]);
   drawnow;             %update to screen as fast as possible
end
```

In the above solution, the function plot is called on each iteration. This will generate a new graphics object each time, and delete the old one. While this works, it is very ineffecient. Using a handle to a single graphics object, we can chose to update on each iteration, instead of making a new one:

```
t = linspace(-pi,+pi,700);
x = cos(t);
y = sin(2*t);

hDot = plot(x(1),y(1), 'o');
xlim([-1.5,1.5]);
ylim([-1.5,1.5]);

for p = 2:length(x)
   set(hDot,'XData',x(p), ...
            'YData',y(p));
   drawnow;
end
```

If you check the documentation on `plot`, you will see that it returns a handle to a graphical object called 'lineseries'. Once you have the name of the object, you can find documentation on it by searching for 'lineseries properties'. Among those are listed 'XData' and 'YData'. The above animation ran faster with the second method. Lets quantify this speed increase. The function `TimedAnimation.m` is provided to do this, and you should make sure you understand the function completely. `TimedAnimation.m` uses the two handy functions `tic` and `toc` for timing, and also shows how you can define and use local functions in your m-files.

# 3 2D functions - $\mathbb{R}^2 \to \mathbb{R}^1$

1D functions ($\mathbb{R} \to \mathbb{R}$), such as $\sin(t)$, are easy to plot. For a parabola, $t^2$:

```
t  = linspace(-2,2,80);
t2 = t.^2;
plot(t,t2);
```

Here, `t` and `t2` are both vectors. Lets say that we wish to plot a 2D parabola: $x^2 + y^2$. The `ndgrid` function allows us to do this:

```
t  = linspace(-2,2,80);
[y,x] = ndgrid(t);
x2 = x.^2;
y2 = y.^2;
f  = x2+y2;
surf(y,x,f);
```

The matrices `x` and `y` stores the x and y coordinates. Elementwise operations are performed in exactly the same way as with vectors, and so `x2`, `y2` and `f` are all matrices.

The `ndgrid` function generates matrices that have variation in one dimension only. The simplest example is the following:

```
[y,x] = ndgrid(1:3)
```

We have already seen and used Matlab boolean logic expressions, such as:

```
1<2
1>2
1==1
1>=1
```

All such expressions are automatically done elementwise, if we supply vectors instead of scalars:

```
  %make a step function:
t  = -10:10;
binarySignal = t>0;      %binary signal, each element either 1 or 0
stem(t, binarySignal);
ylim([-0.5, 1.5])
title('t>0')
```

... and for matrices:

```
[y,x] = ndgrid(t);

binIm1 = x>0;           %binary image, each element either 1 or 0
binIm2 = y>0;

subplot(1,2,1)
imagesc2(t,t,binIm1);
title('x>0')

subplot(1,2,2)
imagesc2(t,t,binIm2);
title('y>0')
```

We can let $x$ and $y$ vary in the interval $[-1, +1]$, using:

```
res   = 40;
t     = linspace(-1,1,res);
[y,x] = ndgrid(t);
```

This will make any resulting matrix be of size `res`, which is the resolution of the image synthesis we are working with[4].

To generate a filled square with one corner at the origin, of some width `w`:

```
w  = 0.5;
binIm1 = (x>=0) & (x<w) & (y>=0) & (y<w);

imagesc2(t,t,binIm1);
title('a square');
```

A disk of radius $r$, is defined by

$$x^2 + y^2 < r^2$$

This is implemented in matlab code as follows:

```
r   = 0.5;        %radius
res = 400;        %resolution

t     = linspace(-1,1,res);   %make a vector
[y,x] = ndgrid(t);            %use the vector to make matrices

binIm =  x.^2 + y.^2  < r^2;  %make the image

imagesc2(t,t,binIm);
```

For a disk(a filled circle) of radius $r$, centered at $(p_x, p_y)$:

$$(x - p_x)^2 + (y - p_y)^2 < r^2$$

Which in Matlab could look like:

```
p = [0.8 0.3];
binIm =  (x-p(1)).^2 + (y-p(2)).^2  < r^2;
imagesc2(t,t,binIm);
```

[4] compare this with the sampling frequency used for our audio examples of previous exercises

## Tasks

Solutions that do not fill the following requirements **EXACTLY, PRECISELY AND TO THE LETTER** will not be considered:

- Send your solutions by email to:

  stefan.karlsson@hh.se

  subject: Matlab, Exercise `X`, `YourNames`

- Send all the files that are requested, no more and no less, in one single zip file per exercise, with NO sub-folders in the zip file. All the files, for all the tasks should be bundled into one zip file.

- Put the Names of the authors, in remarks, at the top of every m-file.

- Send the solutions within 2 weeks of every exercise session. That is, you have a two week deadline to hand it in.

## Task 1

Function `e3_1(rad, res)` makes a disk with radius `rad` move in the same pattern as in task 1 of exercise 2. It moves over an image that is of width and height equal to `res`. The size of the disk should **not** change as the `res` changes. The axis range of the image should be (-1,1) for both x and y.

In order to pass this task you MUST use handles with the `set` function. The key property to set is called 'CData'.

## Hints

- your function could start out like this:

```
function e3_1(rad, res)

t = linspace(-pi,+pi,300);    %time vector for the motion
px = 0.7*cos(t);              %x coordinates of the centre
py = 0.7*sin(2*t);            %y coordinates of the centre

%fill in the rest, it includes making a "binIm", and updating in a for loop
```

## Task 2

Function `e3_2(rad, res)` makes 3 disks with radius `rad`. Each disk should be centered at distinct places in the image. The size of the disks should **not** change as the `res` changes. The axis range of the image should be (-1,1) for both x and y.

The image should be RGB, and each disk should occupy a different channel. One disk each in R, G, and B.

## Hints

- to get distinct points, here is a good suggestion:

```
t = linspace(-pi,+pi,4)+pi/12;
t = t(1:3);
px = 0.7*cos(t);
py = 0.7*sin(t);
```

- the function `imagesc2` handles RGB images

## Task 3 (optional for grade 4)

**You only need to finish this Task if you are aiming for a grade 4 on this exercise!**

The script `e3_3` makes 3 disks with varying radius by calling function `e3_2`. The first frame of the animation should have `rad=0.05`, and the final frame should have `rad=1.5`, while `res=200`. The script should record video of the figures appearance, and save it to file (to myVideoOutput.avi). The video that you save, should display the appearance of the image as the disks grow. The video should contain 100 frames(each frame with a different `rad`).

In order to pass this task, you must yourselves find out how to record video in Matlab. Use documentation and google to find help.

## Hints

Google: 'make avi video matlab'